```
/* getopt.c

   Copyright (C) 1989, 1990, 1991, 1992 Free Software Foundation, Inc.

   This file is part of GNU MCSim.

   GNU MCSim  is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License
   as published by the Free Software Foundation; either version 3
   of the License, or (at your option) any later version.

   GNU MCSim is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with GNU MCSim; if not, see <http://www.gnu.org/licenses/>

   -- Revisions -----
     Logfile:  %F%
    Revision:  %I%
        Date:  %G%
     Modtime:  %U%
      Author:  @a
   -- SCCS  ---------

*/
#include <stdio.h>
#include <stdlib.h>

/* This version of `_getopt' appears to the caller like standard Unix
`_getopt'
   but it behaves differently for the user, since it allows the user
   to intersperse the options with the other arguments.

   As `_getopt' works, it permutes the elements of ARGV so that,
   when it is done, all the options precede everything else.  Thus
   all application programs are extended to handle flexible argument
order.

   Setting the environment variable POSIXLY_CORRECT disables permutation.
   Then the behavior is completely standard.

   Application programs can use a third alternative mode in which
   they can distinguish the relative order of options and other
arguments.
*/

#include "getopt.h"
#include "lexerr.h"

/* For communication from `_getopt' to the caller.
   When `_getopt' finds an option that takes an argument,
```

```
   the argument value is returned here.
   Also, when `ordering' is RETURN_IN_ORDER,
   each non-option ARGV-element is returned here.
*/

char *optarg = 0;

/* Index in ARGV of the next element to be scanned.
   This is used for communication to and from the caller
   and for communication between successive calls to `_getopt'.

   On entry to `_getopt', zero means this is the first call; initialize.

   When `_getopt' returns EOF, this is the index of the first of the
   non-option elements that the caller should itself scan.

   Otherwise, `optind' communicates from one call to the next
   how much of ARGV has been scanned so far.
*/

int optind = 0;

/* The next char to be scanned in the option-element
   in which the last option character we returned was found.
   This allows us to pick up the scan where we left off.

   If this is zero, or a null string, it means resume the scan
   by advancing to the next ARGV-element.
*/

static char *nextchar;

/* Callers store zero here to inhibit the error message
   for unrecognized options.
*/

int opterr = 1;

/* Describe how to deal with options that follow non-option ARGV-
elements.

   If the caller did not specify anything,
   the default is REQUIRE_ORDER if the environment variable
   POSIXLY_CORRECT is defined, PERMUTE otherwise.

   REQUIRE_ORDER means don't recognize them as options;
   stop option processing when the first non-option is seen.
   This is what Unix does.
   This mode of operation is selected by either setting the environment
   variable POSIXLY_CORRECT, or using `+' as the first character
   of the list of option characters.

   PERMUTE is the default.  We permute the contents of ARGV as we scan,
```

```
   so that eventually all the non-options are at the end.  This allows
options
   to be given in any order, even with programs that were not written to
   expect this.

   RETURN_IN_ORDER is an option available to programs that were written
   to expect options and other ARGV-elements in any order and that care
about
   the ordering of the two.  We describe each non-option ARGV-element
   as if it were the argument of an option with character code 1.
   Using `-' as the first character of the list of option characters
   selects this mode of operation.

   The special argument `--' forces an end of option-scanning regardless
   of the value of `ordering'.  In the case of RETURN_IN_ORDER, only
   `--' can cause `_getopt' to return EOF with `optind' != ARGC.
*/

static enum
{
  REQUIRE_ORDER, PERMUTE, RETURN_IN_ORDER
} ordering;

#include <string.h>

/* Avoid depending on library functions or files
   whose names are inconsistent.
*/

/* char *getenv (); FB: this is not a proper declaration, getenv is
already
   in stdlib anyway */

static char *
my_index (char *string, int chr)
{
  while (*string) {
   if (*string == chr) return string;
   string++;
  }
  return 0;
}

static void
my_bcopy (char *from, char *to, int size)
{
  int i;
  for (i = 0; i < size; i++) to[i] = from[i];
}

/* Handle permutation of arguments.  */

/* Describe the part of ARGV that contains non-options that have
```

```
   been skipped.  `first_nonopt' is the index in ARGV of the first of
them;
   `last_nonopt' is the index after the last of them.
*/

static int first_nonopt;
static int last_nonopt;

/* Exchange two adjacent subsequences of ARGV.
   One subsequence is elements [first_nonopt,last_nonopt)
   which contains all the non-options that have been skipped so far.
   The other is elements [last_nonopt,optind), which contains all
   the options processed since those non-options were skipped.

   `first_nonopt' and `last_nonopt' are relocated so that they describe
   the new indices of the non-options in ARGV after they are moved.
*/

static void exchange (char **argv)
{
  int nonopts_size = (last_nonopt - first_nonopt) * sizeof (char *);
  char **temp;

  if (nonopts_size == 0) {
    printf ("Error: zero length array allocation in exchange -
Exiting\n");
    exit (0);
  }

  temp = (char **) malloc (nonopts_size);

  if (temp == NULL)
    ReportError (NULL, RE_OUTOFMEM | RE_FATAL, "exchange", NULL);


  /* Interchange the two blocks of data in ARGV.  */

  my_bcopy (&argv[first_nonopt][0], temp[0], nonopts_size);
  my_bcopy (&argv[last_nonopt][0], &argv[first_nonopt][0],
            (optind - last_nonopt) * sizeof (char *));
  my_bcopy (temp[0], &argv[first_nonopt + optind - last_nonopt][0],
            nonopts_size);

  /* Update records for the slots the non-options now occupy.  */

  first_nonopt += (optind - last_nonopt);
  last_nonopt = optind;
}

/* Scan elements of ARGV (whose length is ARGC) for option characters
   given in OPTSTRING.

   If an element of ARGV starts with '-', and is not exactly "-" or "--",
   then it is an option element.  The characters of this element
```

```
   (aside from the initial '-') are option characters.  If `_getopt'
   is called repeatedly, it returns successively each of the option
characters
   from each of the option elements.

   If `_getopt' finds another option character, it returns that
character,
   updating `optind' and `nextchar' so that the next call to `_getopt'
can
   resume the scan with the following option character or ARGV-element.

   If there are no more option characters, `_getopt' returns `EOF'.
   Then `optind' is the index in ARGV of the first ARGV-element
   that is not an option.  (The ARGV-elements have been permuted
   so that those that are not options now come last.)

   OPTSTRING is a string containing the legitimate option characters.
   If an option character is seen that is not listed in OPTSTRING,
   return '?' after printing an error message.  If you set `opterr' to
   zero, the error message is suppressed but we still return '?'.

   If a char in OPTSTRING is followed by a colon, that means it wants an
arg,
   so the following text in the same ARGV-element, or the text of the
following
   ARGV-element, is returned in `optarg'.  Two colons mean an option that
   wants an optional arg; if there is text in the current ARGV-element,
   it is returned in `optarg', otherwise `optarg' is set to zero.

   If OPTSTRING starts with `-' or `+', it requests different methods of
   handling the non-option ARGV-elements.
   See the comments about RETURN_IN_ORDER and REQUIRE_ORDER, above.

   Long-named options begin with `--' instead of `-'.
   Their names may be abbreviated as long as the abbreviation is unique
   or is an exact match for some defined option.  If they have an
   argument, it follows the option name in the same ARGV-element,
separated
   from the option name by a `=', or else the in next ARGV-element.
   When `_getopt' finds a long-named option, it returns 0 if that
option's
   `flag' field is nonzero, the value of the option's `val' field
   if the `flag' field is zero.

   The elements of ARGV aren't really const, because we permute them.
   But we pretend they're const in the prototype to be compatible
   with other systems.

   LONGOPTS is a vector of `struct option' terminated by an
   element containing a name which is zero.

   LONGIND returns the index in LONGOPT of the long-named option found.
   It is only valid when a long-named option has been found by the most
   recent call.
```

```
   If LONG_ONLY is nonzero, '-' as well as '--' can introduce
   long-named options.
*/

int _getopt_internal (int argc,
                      char *const *argv,
                      const char *optstring,
                      const struct option *longopts,
                      int *longind,
                      int long_only)
{
  int option_index;

  optarg = 0;

  /* Initialize the internal data when the first call is made.
     Start processing options with ARGV-element 1 (since ARGV-element 0
     is the program name); the sequence of previously skipped
     non-option ARGV-elements is empty.  */

  if (optind == 0) {
    first_nonopt = last_nonopt = optind = 1;

    nextchar = NULL;

    /* Determine how to handle the ordering of options and nonoptions.
*/

    if (optstring[0] == '-') {
      ordering = RETURN_IN_ORDER;
      ++optstring;
    }
    else if (optstring[0] == '+') {
      ordering = REQUIRE_ORDER;
      ++optstring;
    }
    else if (getenv ("POSIXLY_CORRECT") != NULL)
      ordering = REQUIRE_ORDER;
    else
      ordering = PERMUTE;
  }

  if (nextchar == NULL || *nextchar == '\0') {
    if (ordering == PERMUTE) {

      /* If we have just processed some options following some non-
options,
         exchange them so that the options come first.  */

      if (first_nonopt != last_nonopt && last_nonopt != optind)
        exchange ((char **) argv);
      else if (last_nonopt != optind)
        first_nonopt = optind;
```

```
      /* Now skip any additional non-options
         and extend the range of non-options previously skipped.  */

      while (optind < argc
             && (argv[optind][0] != '-' || argv[optind][1] == '\0'))
        optind++;

      last_nonopt = optind;
    }

    /* Special ARGV-element `--' means premature end of options.
       Skip it like a null option,
       then exchange with previous non-options as if it were an option,
       then skip everything else like a non-option.  */

    if (optind != argc && !strcmp (argv[optind], "--")) {
      optind++;

      if (first_nonopt != last_nonopt && last_nonopt != optind)
        exchange ((char **) argv);
      else if (first_nonopt == last_nonopt)
        first_nonopt = optind;

      last_nonopt = argc;

      optind = argc;
    }

    /* If we have done all the ARGV-elements, stop the scan
       and back over any non-options that we skipped and permuted.  */

    if (optind == argc) {
      /* Set the next-arg-index to point at the non-options
         that we previously skipped, so the caller will digest them.  */
      if (first_nonopt != last_nonopt) optind = first_nonopt;
      return EOF;
    }

    /* If we have come to a non-option and did not permute it,
       either stop the scan or describe it to the caller and pass it by.
*/

    if ((argv[optind][0] != '-' || argv[optind][1] == '\0')) {
      if (ordering == REQUIRE_ORDER) return EOF;
      optarg = argv[optind++];
      return 1;
    }

    /* We have found another option-ARGV-element.
       Start decoding its characters.  */

    nextchar = (argv[optind] + 1 +
                (longopts != NULL && argv[optind][1] == '-'));
```

```
}

if (longopts != NULL
    && ((argv[optind][0] == '-'
    && (argv[optind][1] == '-' || long_only)))) {
  const struct option *p;
  char *s = nextchar;
  int exact = 0;
  int ambig = 0;
  const struct option *pfound = NULL;
  int indfound=0;  /* Zeng initialized Jan 19 94 */

  while (*s && *s != '=') s++;

  /* Test all options for either exact match or abbreviated matches. */
  for (p = longopts, option_index = 0; p->name; p++, option_index++)
    if (!strncmp (p->name, nextchar, s - nextchar)) {
      if (s - nextchar == strlen (p->name)) {
        /* Exact match found.  */
        pfound = p;
        indfound = option_index;
        exact = 1;
        break;
      }
      else
        if (pfound == NULL) {
          /* First nonexact match found.  */
          pfound = p;
          indfound = option_index;
        }
        else
          /* Second nonexact match found.  */
          ambig = 1;
    }

  if (ambig && !exact) {
    if (opterr)
      printf ("%s: option `%s' is ambiguous\n",
              argv[0], argv[optind]);
    nextchar += strlen (nextchar);
    optind++;
    return '?';
  }

  if (pfound != NULL) {
    option_index = indfound;
    optind++;
    if (*s) {
      /* Don't test has_arg with >, because some C compilers don't
         allow it to be used on enums. */
        if (pfound->has_arg)
      optarg = s + 1;
      else {
        if (opterr) {
```

```
        if (argv[optind - 1][1] == '-')
          /* --option */
          printf ("%s: option `--%s' doesn't allow an argument\n",
                  argv[0], pfound->name);
        else
          /* +option or -option */
          printf ("%s: option `%c%s' doesn't allow an argument\n",
                  argv[0], argv[optind - 1][0], pfound->name);
      }
      nextchar += strlen (nextchar);
      return '?';
    }
  }
  else
    if (pfound->has_arg == 1) {
      if (optind < argc)
        optarg = argv[optind++];
      else {
        if (opterr)
          printf ("%s: option `%s' requires an argument\n",
                  argv[0], argv[optind - 1]);
        nextchar += strlen (nextchar);
        return '?';
      }
    }

  nextchar += strlen (nextchar);

  if (longind != NULL) *longind = option_index;

  if (pfound->flag) {
    *(pfound->flag) = pfound->val;
    return 0;
  }
  return pfound->val;
}

/* Can't find it as a long option.  If this is not _getopt_long_only,
   or the option starts with '--' or is not a valid short
   option, then it's an error.
   Otherwise interpret it as a short option. */
if (!long_only || argv[optind][1] == '-'
    || my_index ((char *) optstring, *nextchar) == NULL) {
  if (opterr) {
    if (argv[optind][1] == '-')
      /* --option */
      printf ("%s: unrecognized option `--%s'\n",
              argv[0], nextchar);
    else
      /* +option or -option */
      printf ("%s: unrecognized option `%c%s'\n",
              argv[0], argv[optind][0], nextchar);
  }
  nextchar += strlen (nextchar);
```

```
    optind++;
    return '?';
  }
}

/* Look at and handle the next option-character. */

{
  char c = *nextchar++;
  char *temp = my_index ((char *) optstring, c);

  /* Increment `optind' when we start to process its last character. */
  if (*nextchar == '\0') optind++;

  if (temp == NULL || c == ':') {
    if (opterr) {
      if (c < 040 || c >= 0177)
        printf ("%s: unrecognized option, character code 0%o\n",
                argv[0], c);
      else
        printf ("%s: unrecognized option `-%c'\n", argv[0], c);
    }
    return '?';
  }

  if (temp[1] == ':') {
    if (temp[2] == ':') {
      /* This is an option that accepts an argument optionally. */
      if (*nextchar != '\0') {
        optarg = nextchar;
        optind++;
      }
      else optarg = 0;
      nextchar = NULL;
    }
    else {
      /* This is an option that requires an argument. */
      if (*nextchar != 0) {
        optarg = nextchar;
        /* If we end this ARGV-element by taking the rest as an arg,
           we must advance to the next element now.  */
        optind++;
      }
      else
        if (optind == argc) {
          if (opterr)
            printf ("%s: option `-%c' requires an argument\n",
                    argv[0], c);
          c = '?';
        }
        else
          /* We already incremented `optind' once;
          increment it again when taking next ARGV-elt as argument.  */
```

```
          optarg = argv[optind++];
          nextchar = NULL;
        }
      }

      return c;
    }
}

int _getopt (int argc, char *const *argv, const char *optstring)
{
  return _getopt_internal (argc, argv, optstring,
                           (const struct option *) 0,
                           (int *) 0, 0);
}

#ifdef TEST

/* Compile with -DTEST to make an executable for use in testing
   the above definition of `_getopt'. */

int
main (argc, argv)
     int argc;
     char **argv;
{
  int c;
  int digit_optind = 0;

  while (1) {
    int this_option_optind = optind ? optind : 1;

    c = _getopt (argc, argv, "abc:d:0123456789");
    if (c == EOF)
      break;

    switch (c) {
      case '0':
      case '1':
      case '2':
      case '3':
      case '4':
      case '5':
      case '6':
      case '7':
      case '8':
      case '9':
        if (digit_optind != 0 && digit_optind != this_option_optind)
          printf ("digits occur in two different argv-elements.\n");
        digit_optind = this_option_optind;
        printf ("option %c\n", c);
        break;

      case 'a':
```

```
        printf ("option a\n");
        break;

      case 'b':
        printf ("option b\n");
        break;

      case 'c':
        printf ("option c with value `%s'\n", optarg);
        break;

      case '?':
        break;

      default:
        printf ("?? _getopt returned character code 0%o ??\n", c);
    }
  }

  if (optind < argc) {
    printf ("non-option ARGV-elements: ");
    while (optind < argc)
      printf ("%s ", argv[optind++]);
    printf ("\n");
  }

  exit (0);
}

#endif /* TEST */
```